120 6-9-04 #8 #F# 6-16-4 2127

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE
BEFORE THE BOARD OF PATENT APPEALS AND INTERFERENCES

Express Mail No.: EV332851149US

In re Application of:	Harry Beatty et al.	Examiner:	Syed J. Ali
Serial No.:	09/597,524	Group Art Unit:	2127
Filing Date	June 20, 2000	Date:	June 8, 2004

Title **METHOD OF USING A DISTINCT FLOW OF COMPUTATIONAL CONTROL AS A REUSABLE ABSTRACT DATA OBJECT**

Mail Stop Appeal Brief - Patents
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

I hereby certify that this paper or fee is being deposited with the United States Postal Service "Express Mail Post Office to Addressee" service under 37 CFR 1.10 on the date indicated above and is addressed to Mail Stop Appeal Brief – Patents, Commissioner for Patents, P.O. box 1450, Alexandria, VA 22313-1450

Name: Barbara Browne Date: June 8, 2004

Signature Barbara Browne

BRIEF FOR APPELLANTS

This is an appeal from the Final Rejection of the Examiner mailed July 31, 2003, rejecting claims 1-16. A Notice of Appeal and the Appeal Fee were timely mailed April 8, 2004. Please charge the Appeal fee of $ 330.00 for this brief and any over or under payment to Deposit Account No. 09-0458, of the assignee International Business Machines Corporation. Three copies of the brief are enclosed.

REAL PARTY IN INTEREST

The real party in interest is the assignee of all rights in this application, International Business Machines Corporation, a corporation of the State of New York, having a place of business at Armonk, New York 10504.

RELATED APPEALS AND INTERFERENCES

There are no appeals or interferences known to appellants, appellants' legal representatives or assignee, which will directly affect or be affected by, or have a bearing on the Board's decision on this appeal.

STATUS OF CLAIMS

The subject application was filed on June 20, 2000 with claims 1-16. An Amendment was filed on October 31, 2003, responsive to the Office Action mailed July 31, 2003, amending claim 9. The Final Rejection Office Action was mailed January 14, 2004, finally rejecting all claims in the application, to wit, claims 1-16. Appellants are appealing the rejection of these claims.

STATUS OF AMENDMENTS

All the amendments added during prosecution of the application have been entered and are presently in the application. The rejected claims 1-16 as they presently stand are set forth in the Appendix. A summary of the rejection of the claims may be found in the Final Rejection Office Action mailed January 14, 2004.

SUMMARY OF THE INVENTION

Appellants' invention is directed to a method of parallel computer processing. Parallel processing in general offers improvements in that a single program can run simultaneously different "threads" or independent flows of control managed by the program. In the prior art, each thread is assigned a specific unit of work to perform, generally in parallel, and when the work is finished, the threads cease to exist. There is a cost to create, manage and terminate a thread in both machine-cycle components and programming complexity components. The prior art's use of threads treats the threads and data differently, which creates complexity and resulting error-proneness during implementation. Specification, p.3, ll.7-16.

Unlike the prior art, which treats contexts and data differently, the present invention allows a thread (or flow of control) to be treated as a data object by the software, so that threads may be created once and reused as needed. Specification, p.19, ll.18-23. In, particular, the method of the present invention (claims 1-10) uses both first and second threads, with the first thread having two states: 1) a first state processing work for the program structure and 2) a second state undispatched awaiting work to process. Specification, p.20, ll.15-25; Fig. 11 "First Thread Bottle Born Locked." In the method of the present invention, the second thread is used to prepare work for the first thread to process, which then places the work prepared by the second thread in a queue for processing by the first thread. Specification, p.21, ll.7-13; Fig. 11 "Second Thread Launcher." If the first thread is awaiting work to process when the work prepared by the second thread is placed in the queue, the first thread is dispatched and used to process the

work in the queue. Specification, p.20, l.25 to p.21, l.1; Fig. 11 "Any Work Items? – Yes – Do Work." If the first thread is processing other work when the work prepared by the second thread is placed in the queue, the first thread is used to complete processing of the other work, access the work in the queue, and then process the work in the queue. Specification, p.21, ll.1-4; Fig. 11 "First Thread Bottle Born Locked."

The second thread may continue to place additional work in the queue, and the first thread sequentially processes the additional work in the queue as it completes processing prior work (method claims 2 and 7). Specification, p.21, ll.1-4 and 10-13; Fig. 11 "First Thread Bottle Born Locked" and "Second Thread Launcher." Preferably, the second thread marks the work placed in the first thread queue as not complete (method claims 3 and 7). Specification, p.21, ll.13-14. If the first thread is processing other work when the work prepared by the second thread is placed in the queue, and when the first thread completes processing of the work in the queue, the method may include using the first thread to mark the completed work as complete. Specification, p.21, ll.1-2. Subsequent work from the second thread is made to wait until the previous work in the first thread is marked complete (method claims 4 and 8). Specification, p.21, ll.2-4. The first thread may be reused to process other work (method claims 5 and 9), and the program structure may destroy the first thread after it completes a desired amount of work (method claims 6 and 10). Specification, p.21, ll.17-19.

The program storage device (claims 11-16) defines a machine that tangibly embodies a program of instructions to perform the aforementioned method of parallel processing. Specification, p.21, ll.20-24.

ISSUES

The issue in this appeal is whether claims 1-16 are obvious to one of ordinary skill in the art under 35 USC § 103(a) from the cited prior art.

Claims 1-5, 7-9 and 11-15 stand rejected under 35 USC § 103(a) as being obvious from Achenson et al. U.S. Patent No. 6,477,586 in view of LiVecchi U.S. Patent No. 6,427,161.

Claims 6, 10 and 16 stand rejected under 35 USC § 103(a) as being obvious from Achenson in view of LiVecchi in view of Voll et al U.S. Patent No. 6,170,018.

GROUPING OF THE CLAIMS

The following claim groups stand or fall together:

Claim Group A: Claims 1 and 11

Claim Group B: Claims 2 and 12

Claim Group C: Claims 3 and 13

Claim Group D: Claims 4 and 14

Claim Group E: Claims 5, 9 and 15

Claim Group F: Claims 6, 10 and 16

Claim Group G: Claim 7

Claim Group H: Claim 8

ARGUMENT

I. Prior Art

U.S. Patent No. 6,477,586 to Achenson et al. discloses a multi threaded, multi processed distributed system in which different processes use a dispatcher thread for passing a remote call procedure (RCP) message to an appropriate available thread from a pool of worker threads within the process, to permit the RCP message to be processed. Achenson discloses nothing about a thread having two states, or using a second thread to prepare work and place it in a queue for processing by the first thread when it is completed processing any other work.

U.S. Patent No. 6,427,161 to LiVecchi is directed to a computer server system that supports concurrent execution by multiple threads, and in particular, to use of a dispatcher thread that is typically responsible for monitoring the queue which receives incoming connection requests for a passive socket. The dispatcher thread is distinguished from those threads to which the work is dispatched, which are called the "worker" threads. LiVecchi, column 3, lines 15-21. LiVecchi describes one implementation where a separate dispatcher thread keeps track of the status of each worker thread and, only when the worker thread is in an idle state and has no work currently assigned to it, will the dispatcher thread assign an incoming request to that worker thread. LiVecchi, column 3, lines 15-31. LiVecchi states that, "the dispatcher thread may become a bottleneck that prevents worker threads from being scheduled fast enough to keep all of the processors busy." LiVecchi, column 3, lines 29-31. LiVecchi solves this bottlenecking problem by the alternative system described at column 3, lines 32-50. This approach is "implemented

without using a dispatcher thread." LiVecchi, column 3, lines 32-33 (emphasis added). Instead the worker threads are themselves responsible for checking a passive socket queue to determine if there are any connection requests, and if a request is waiting the thread removes the request from the queue and begins to process it. If no request is waiting, the thread then becomes an idle thread, which then sleeps.

U.S. Patent No. 6,170,018 to Voll et al is directed to a remote calling procedure between computer processes.

II. The Examiner's Rejections and Appellants' Arguments

Claim Group A: Claims 1 and 11

Claims 1 and 11 recite a method of parallel processing utilizing two threads which each represent an independent flow of control managed by separate program structures. The first thread has two states: 1) processing work for the program structure, and 2) undispatched awaiting work to process. The method involves using the second thread to prepare work for the first thread to process and placing the work in a queue. If the first thread is awaiting work to process when the work is placed in the queue, the first thread is dispatched and processes the work in the queue. If the first thread is processing other work when the second thread place the work in the queue, the first thread completes processing of the other work then accesses the work and processes it from the queue.

The hypothetical combinations of Achenson and LiVecchi as cited does not present a prima facie case of obviousness to one of ordinary skill in the art. As the Examiner has recognized, Achenson discloses nothing about a thread having two states, and discloses nothing about using a second thread to prepare work and place it in a queue for

processing by the first thread when it is completed processing any other work. Jan. 14, 2004 Office Action, p.4.

The Examiner cites the LiVecchi patent for this missing disclosure. However, LiVecchi at column 3, lines 15-31 describes in a first embodiment one implementation where a separate dispatcher thread keeps track of the status of each worker thread and, only when the worker thread is in an idle state and has no work currently assigned to it, will the dispatcher thread assign an incoming request to that worker thread. As LiVecchi recognizes, "the dispatcher thread may become a bottleneck that prevents worker threads from being scheduled fast enough to keep all of the processors busy." LiVecchi, column 3, lines 29-31. LiVecchi solves this bottlenecking problem by a second, alternative embodiment which is "implemented without using a dispatcher thread." LiVecchi, column 3, lines 32-33 (emphasis added). Instead the worker threads are themselves responsible for checking a passive socket queue to determine if there are any connection requests, and if a request is waiting the thread removes the request from the queue and begins to process it. If no request is waiting, the thread then becomes an idle thread which then sleeps. Thus, in this latter system cited by the Examiner where the worker threads have two states, it is specifically disclosed that no dispatcher thread is utilized.

An invention may not be considered to be obvious if it fails to consider a reference in its entirety and ignores portions that teach away from the invention. *Bausch & Lomb, Inc. v. Barnes-Hind/Hydrocurve, Inc.*, 796 F.2d 443, 230 USPQ 416, 420 (Fed. Cir. 1986). *See also, Tec Air Inc. v. Denso Mfg. Michigan Inc.*, 192 F.3d 1353, 52 USPQ2d 1294, 1298 (Fed. Cir. 1999) ("There is no suggestion to combine, however, if a reference teaches

away from its combination with another source.") Here, LiVecchi himself teaches away from utilizing a second thread to prepare work for a first thread that has two states and, instead, utilizes the two-state thread without a second or dispatcher thread. The Examiner attempts to avoid this contradiction by taking the position that "[a]lthough LiVecchi does not use a dispatching thread for preparing work, this aspect of the claimed invention comes from Achenson rather than LiVecchi" Jan. 14, 2004 Office Action, p.8. The primary reference (Achenson) does use a dispatcher/worker thread combination. However, when the secondary reference (LiVecchi) discloses a first embodiment of a dispatcher/worker thread combination, where the worker thread has only one state, and an alternative second embodiment of a two state worker thread that intentionally uses no dispatcher, one of ordinary skill in the art would naturally combine Achenson's dispatcher/worker thread combination with LiVecchi's similar first embodiment of a dispatcher/worker thread combination.

In order to combine Achenson with LiVecchi's second embodiment, which teaches away from using a dispatcher thread, there would have to be some different motivation taught in one of the references. *Winner International Royalty Corp. v. Wang*, 202 F.3d 1340, 53 USPQ2d 1580, 1587 (Fed. Cir. 2000) ("Trade offs often concern what is feasible, not what is, on balance, desirable. Motivation to combine requires the latter.") However, the Examiner has cited no teaching or motivation for one of ordinary skill in the art to disregard one or the other reference's teaching to combine them in the manner proposed. Thus the present invention as defined by appellants' claim 1 is not prima facie obvious since the hypothetical combination of elements, chosen as a result of the hindsight benefit

of reading appellants' own specification, does not arrive at the present invention as described in claims 1 and 11.

Claim Group B: Claims 2 and 12

Dependent claims 2 and 12 (dependent on independent claims 1 and 11, respectively) recite that the second thread continues to place additional work in the queue, which is sequentially processed by the first thread as it completes processing prior work. The disclosure cited by the Examiner in Achenson at column 6, lines 50-54 does not make up for the teaching in LiVecchi that a two-state thread, i.e., appellants' first thread, is not used with a second or dispatcher thread. Moreover, LiVecchi states that "[w]hen work is assigned to an idle thread, it is no longer considered idle, and no further work will be assigned to it until it has completed its current work request." LiVecchi, column 3, lines 26-28. Unlike LiVecchi, which stops assigning work to a busy "worker" thread, appellants' invention defined by claims 2 and 12 has the second thread continue to assign and place work in the queue for the first thread, even if the first thread is busy. Thus, LiVecchi again contradicts Achenson, so that the hypothetical combination of Achenson and LiVecchi does not suggest this claim limitation.

Claim Group C: Claims 3 and 13

Appellants' claims 3 and 13, dependent on independent claims 1 and 11, respectively, recite that the second thread marks the work placed in the first thread queue as not complete. Again, the combination of Achenson and LiVecchi is not properly made since LiVecchi teaches away from utilizing a second dispatcher thread with a first two-state thread. Nevertheless, neither Achenson nor LiVecchi alone or in combination suggest this

limitation. The Examiner's quotation from Achenson at column 6, lines 42-43, referencing setting the hConn value of the RPC request to null, does not disclose or suggest marking work placed in a queue for a worker thread as not complete, since when the hConn value is later updated, the work has still not been completed by the worker thread. Achenson, column 6, lines 48-50. Further, LiVecchi, which uses a dispatcher thread and worker thread combination in one embodiment, and a busy/idle thread in a different embodiment, has no need to mark work as "not complete." In LiVecchi's first embodiment, the dispatcher thread assigns no work to a worker thread that is not idle, and in his second embodiment there is no thread that assigns or marks work to be done by the worker thread. Thus, claims 3 and 13 are not obvious to one of ordinary skill in the art from the Examiner's proposed combination.

Claim Group D: Claims 4 and 14

Dependent claims 4 and 14, dependent on independent claims 1 and 11, respectively, recite that work placed in the first thread's queue by the second thread is made to wait until completed work is marked complete by the first thread. Again, there is no suggestion to combine Achenson and LiVecchi for the reasons given above and, even if combined, they do not suggest such marking. The quote from Achenson at column 6, lines 48-50 does not suggest that the work has been marked complete since, at that point, the work has still not yet been assigned by the dispatcher thread to the appropriate worker thread. Achenson, column 6, lines 48-54. LiVecchi's first embodiment does not suggest this limitation because a worker thread does not need to mark work complete because there is never a queue containing work to be assigned to it. LiVecchi's second

embodiment does not suggest this limitation because the worker thread goes into idle mode when it completes work. Accordingly, Achenson and LiVecchi in the proposed combination do not present a prima facie case of obviousness of these claims.

Claim Group E: Claims 5, 9 and 15

Claim 5, dependent on claim 1, and claim 15, dependent on claim 11, add that the first thread is reused to process other work. Claim 9 adds this same subject matter, and is dependent on claim 7 (which is itself a combination of the subject matter of claims 1, 2 and 3). For this rejection, the Examiner cites from LiVecchi, "as each thread completes the work requested has been processed, it looks on the queue for its next request" (LiVecchi column 3, lines 35-37). However, this aspect of the various processes disclosed therein is from the embodiment that is "implemented without using a dispatcher thread." LiVecchi column 3, lines 32-33. Accordingly, claims 5, 9 and 15 are allowable because one of ordinary skill in the art would not look to this portion of LiVecchi in connection with appellants' claimed process using a second thread to provide work in a queue to a first, two-state thread.

Claim Group F: Claims 6, 10 and 16

Appellants' dependent claims 6 and 16 (dependent on claims 4 and 14, respectively) recite that the program structure destroys the first thread after it completes a desired amount of work. Claim 10 adds this same subject matter, and is dependent on claim 8 (which is itself a combination of the subject matter of claims 1, 2, 3 and 4). Again, appellant submits that prima facie obviousness is not established by the combination of Achenson and LiVecchi for the reasons described above, and because LiVecchi's teaching

away from the method of the present invention is not rectified by any teaching in Voll. Voll describes no two-state first thread which is fed work in a queue by a second thread. Voll is only cited for its disclosure that a thread may be destroyed when processing by it completes. This disclosure does not suggest appellants' invention wherein the first thread repeatedly cycles between the processing work state and the undispatched awaiting work state. As it receives work from a queue fed by a second thread, after which the first thread is destroyed after it completes a desired amount of work. Accordingly, appellants' invention as recited in claims 6 and 16 is not obvious from any combination of Achenson, LiVecchi and Voll.

Claim Group G: Claim 7

Independent claim 7 combines the subject matter of claims 1, 2 and 3. Claim 7 incorporates into the subject matter of claim 1, the process using a second thread to provide work in a queue to a first, two-state thread, that the second thread marks the work placed in the first thread queue as not complete, and that the second thread continues to place additional work in the queue, which is sequentially processed by the first thread as it completes processing prior work. Accordingly, if claim groups A, B or C are deemed allowable, claim 7 is allowable as well.

As stated above, the hypothetical combination of Achenson and LiVecchi does not suggest this claim limitation. Achenson discloses nothing about a thread having two states. LiVecchi, which uses a dispatcher thread and worker thread combination in one embodiment, and a busy/idle thread in a different embodiment, has no need to mark work as "not complete" and simply stops assigning work to a busy "worker" thread.

Claim Group H: Claim 8

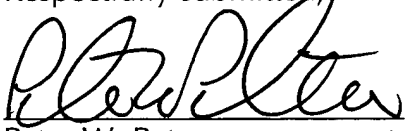
Claim 8, dependent on claim 7, adds the same subject matter as claim 4, namely, that work placed in the first thread's queue by the second thread is made to wait until completed work is marked complete by the first thread. If claim group D is deemed allowable, claim will also be allowable.

The proposed combination of Achenson and LiVecchi do not present a prima facie case of obviousness because in LiVecchi's first embodiment, a worker thread does not need to mark work complete because there is never a queue containing work to be assigned to it, and in LiVecchi's second embodiment, the worker thread goes into idle mode when it completes work.

CONCLUSION

For the reasons given above, appellants submit that the instant application is in condition for allowance. Reversal of the final rejection and passage of the above application to issue are respectfully requested.

Respectfully submitted,



Peter W. Peterson
Reg. No. 31,867

DeLIO & PETERSON, LLC
121 Whitney Avenue
New Haven, CT 06510-1241
(203) 787-0595

APPENDIX

Rejected Claims of Serial No. 09/597,524

1. (original) A method of parallel processing comprising:

providing a first thread which represents an independent flow of control managed by a program structure, said first thread having two states, a first state processing work for the program structure and a second state undispatched awaiting work to process;

providing a second thread which represents an independent flow of control managed by a program structure separate from the first thread;

using the second thread to prepare work for the first thread to process;

placing the work prepared by the second thread in a queue for processing by the first thread;

if the first thread is awaiting work to process when the work prepared by the second thread is placed in the queue, dispatching the first thread and using it to process the work in the queue;

if the first thread is processing other work when the work prepared by the second thread is placed in the queue, using the first thread to complete processing of the other work, access the work in the queue, and then process the work in the queue.

2. (original) The method of claim 1 wherein the second thread continues to place additional work in the queue, and the first thread sequentially processes the additional work in the queue as it completes processing prior work.

3. (original) The method of claim 1 wherein the second thread marks the work placed in the first thread queue as not complete.

4. (original) The method of claim 1 wherein if the first thread is processing other work when the work prepared by the second thread is placed in the queue, and when the first thread completes processing of the work in the queue, using the first thread to mark the completed work as complete, wherein subsequent work from the second thread is made to wait until the previous work in the first thread is marked complete.

5. (original) The method of claim 1 wherein the first thread is reused to process other work.

6. (original) The method of claim 4 wherein the program structure destroys the first thread after it completes a desired amount of work.

7. (original) A method of parallel processing comprising:

providing a first thread which represents an independent flow of control managed by a program structure, said first thread having two states, a first state processing work for the program structure and a second state undispatched awaiting work to process;

providing a second thread which represents an independent flow of control managed by a program structure separate from the first thread;

using the second thread to prepare work for the first thread to process;

placing the work prepared by the second thread in a queue for processing by the first thread, the work placed in the first thread queue being marked as not complete;

if the first thread is awaiting work to process when the work prepared by the second thread is placed in the queue, dispatching the first thread and using it to process the work in the queue;

if the first thread is processing other work when the work prepared by the second thread is placed in the queue, using the first thread to complete processing of the other work, access the work in the queue, and then process the work in the queue;

using the second thread to place additional work in the queue; and

using the first thread to sequentially process the additional work in the queue as it completes processing prior work.

8. (original) The method of claim 7 wherein if the first thread is processing other work when the work prepared by the second thread is placed in the queue, and when the first thread completes processing of the work in the queue, using the first thread to mark the completed work as complete, wherein subsequent work from the second thread is made to wait until the previous work in the first thread is marked complete.

9. (previously presented) The method of claim 7 wherein the first thread is reused to process other work.

10. (original) The method of claim 8 wherein the program structure destroys the first thread after it completes a desired amount of work.

11. (original) A program storage device readable by a machine, tangibly embodying a program of instructions executable by the machine to perform method steps of parallel processing using i) a first thread which represents an independent flow of control managed by a program structure, said first thread having two states, a first state processing work for the program structure and a second state undispatched awaiting work to process, and ii) a second thread which represents an independent flow of control managed by a program structure separate from the first thread, said method steps comprising:

using the second thread to prepare work for the first thread to process;

placing the work prepared by the second thread in a queue for processing by the first thread;

if the first thread is awaiting work to process when the work prepared by the second thread is placed in the queue, dispatching the first thread and using it to process the work in the queue;

if the first thread is processing other work when the work prepared by the second thread is placed in the queue, using the first thread to complete processing of the other work, access the work in the queue, and then process the work in the queue.

12. (original) The program storage device of claim 11 wherein, in the method steps, the second thread continues to place additional work in the queue, and the first thread sequentially processes the additional work in the queue as it completes processing prior work.

13. (original) The program storage device of claim 11 wherein, in the method steps, the second thread marks the work placed in the first thread queue as not complete.

14. (original) The program storage device of claim 11 wherein, in the method steps, if the first thread is processing other work when the work prepared by the second thread is placed in the queue, and when the first thread completes processing of the work in the queue, using the first thread to mark the completed work as complete, wherein subsequent work from the second thread is made to wait until the previous work in the first thread is marked complete.

15. (original) The program storage device of claim 11 wherein, in the method steps, the first thread is reused to process other work.

16. (original) The program storage device of claim 14 wherein, in the method steps, the program structure destroys the first thread after it completes a desired amount of work.